

Fuse MQ Enterprise **Managing and Monitoring a Broker**

Version 7.0
April 2012

Integration Everywhere

Managing and Monitoring a Broker

Version 7.0

Updated: 25 Apr 2012

Copyright © 2011 FuseSource Corp. All rights reserved.

Legal Notices

Progress Software Corporation and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from Progress Software Corporation, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

Progress, IONA, IONA Technologies, the IONA logo, Orbix, High Performance Integration, Artix, Fuse, and Making Software Work Together are trademarks or registered trademarks of Progress Software Corporation and/or its subsidiaries in the US and other countries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the US and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate Progress Software Corporation makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Progress Software Corporation shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

All products or services mentioned in this manual are covered by the trademarks, service marks, or product names as designated by the companies who market those products.

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of Progress Software Corporation. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. Progress Software Corporation assumes no responsibility for errors or omissions contained in this publication. This publication and features described herein are subject to change without notice. Portions of this document may include Apache Foundation documentation, all rights reserved.

Table of Contents

1. Introduction	9
2. Editing a Broker's Configuration	13
Editing a Standalone Broker's Configuration	14
Editing a Broker's Configuration in a Fabric	16
3. Setting up and Accessing the Fuse MQ Enterprise Web Console	21
Using the Embedded Console	22
Deploying a Standalone Console	25
4. Installing Fuse MQ Enterprise as a Service	27
Generating the Service Wrapper	28
Configuring the Wrapper	32
Installing and Starting the Service	36
5. Starting a Broker	39
6. Sending Commands to the Broker	43
7. Shutting Down a Broker	47
Shutting Down a Local Broker	48
Shutting Down a Broker Remotely	49
8. Using Logging	53
Logging Configuration	54
Viewing the Log	56
9. Using JMX	59
Configuring JMX	60
Statistics Collected by JMX	62
Managing the Broker with JMX	65
10. Applying Patches	71
Index	73

List of Tables

4.1. Service Wrapper Installation Options	29
4.2. Wrapper Logging Properties	34
5.1. Start up Commands for Console Mode	39
5.2. Start up Commands for Daemon Mode	40
6.1. Administration Client Arguments	43
9.1. Broker JMX Configuration Properties	61
9.2. Broker JMX Statistics	62
9.3. Destination JMX Statistics	62
9.4. Connection JMX Statistics	63
9.5. Broker MBean Operations	65
9.6. Connector MBean Operations	66
9.7. Network Connector MBean Operations	67
9.8. Queue MBean Operations	67
9.9. Topic MBean Operations	69
9.10. Subscription MBean Operations	70

List of Examples

2.1. Adding Property Placeholder Support to Fuse MQ Enterprise Configuration	18
2.2. Configuraiton with Property Placeholders	18
2.3. Importing an XML Configuration Template	19
2.4. Creating a Profile Using an XML Configuration Template	19
2.5. Creating a Deployment Profile	19
2.6. Setting Properties in a Profile	20
2.7. Setting the Broker Name Property	20
2.8. Assigning a Profile to a Broker	20
3.1. Creating a New Profile	22
3.2. Editing a Profile to Include the Web Console	22
3.3. Adding the Web Console Profile to a Broker	23
3.4. Changing the Web Console's Port	24
3.5. Changing the Web Console's Port in a Fabric	24
3.6. Disabling the Embedded Web Console	25
3.7. Configuration for Deploying the Web Console in Tomcat	26
3.8. Configuration for Monitoring a Cluster with the Web Console	26
4.1. Wrapper Install Help	28
4.2. Generating a Service Wrapper	30
4.3. Default Environment Settings	33
4.4. Default Java System Properties	33
4.5. Default Wrapper Classpath	33
4.6. Wrapper JMX Properties	34
5.1. Broker Console	40
5.2. Starting a Broker in a Fabric	40
5.3. Starting a Broker in a Fabric with the Administration Client	41
6.1. Client Command	43
6.2. Console Help	44
6.3. Help for a Command	45
7.1. Using the Console's Shutdown Command	48
7.2. Stop Command Syntax	49
7.3. Stopping a Remote Broker	50
7.4. Shutting DOWn a Broker using a Remote Console Connection	50
7.5. Shutting Down a Broker in a Fabric	51
8.1. Changing Logging Levels	55
8.2. Changing the Log Information Displayed on the Console	55
9.1. Configuring a Broker's JMX Connection	61

Chapter 1. Introduction

Once a messaging solution is deployed it needs to be monitored to ensure it performs at peak performance. When problems do arise, many of them can be solved using the broker's administrative tools. The broker's administrative tools can also be used to provide important debugging information when troubleshooting problems.

Overview

Message brokers are long lived and usually form the backbone of the applications of which they are a part. Over the course of a broker's life span, there are a number of management tasks that you may need to do to keep the broker running at peak performance. This includes monitoring the health of the broker, adding destinations, and security certificates.

If applications run into trouble one of the first places to look for clues is the broker. The broker is unlikely to be the root cause of the problem, but its logs and metrics will provide clues as to what is the root cause. You may also be able to resolve the problem using the broker's administrative interface.

Routine tasks

While Fuse MQ Enterprise is designed to require a light touch for management, there are a few routine management tasks that need to be performed:

- installing SSL certificates
 - starting the broker
 - creating destinations
 - stopping the broker
 - maintaining the advisory topics
 - monitoring the health of the broker
 - monitoring the health of the destinations
-

Troubleshooting

If an application runs into issues the broker will usually be able to provide clues to what is going wrong. Because the broker is central to the operation of any application that relies on messaging, it will be able to provide clues even if the broker is functioning properly. You may also be able to solve the problem by making adjustments to the broker's configuration.

Common things to check for clues as to the nature of a problem include:

- the broker's log file
- the advisory topics
- the broker's overall memory footprint
- the size of individual destination
- the total number of messages in the broker
- the size of the broker's persistent store
- a thread dump of the broker

One or more of these items can provide information about the problem. For example, if a destination grows to a very large size it could indicate that one of its consumers is having trouble keeping up with the messages. If the broker's log also shows that the consumer is repeatedly connecting and disconnecting from the destination, that could indicate a networking problem or a problem with the machine hosting the consumer.

Tools

There are a number of tools that you can use to monitor and administer Fuse MQ Enterprise.

The following tools are included with Fuse MQ Enterprise:

- [administration client on page 43](#)—a command line tool that can be used to manage a broker and do rudimentary metric reporting
- [console mode on page 44](#)—a runtime mode that presents you with a custom console that provides a number of administrative options
- [Web console on page 21](#)—a browser based console that provides metric reporting, destination browsing, and other administrative functions

FuseSource also provides management tools that you can install as part of your subscription:

- [Fuse Management Console](#)¹—a browser based console for viewing, monitoring, and deploying broker clusters

¹ <http://fusesource.com/docs/fmc>

- [Fuse HQ](#)²—an advanced monitoring and management tool that can provide detailed metrics and alerting.

In addition to the FuseSource supplied tools there are a number of third party tools that can be used to administer and monitor a broker including:

- jconsole—a JMX tool that is shipped with the JDK
- [VisualVM](#)³—a visual tool integrating several command line JDK tools and lightweight profiling capabilities

² <http://fusesource.com/docs/hq>

³ <http://visualvm.java.net/>

Chapter 2. Editing a Broker's Configuration

Fuse MQ Enterprise configuration requires making changes in a number of properties. How the properties are changed depends on how the broker is deployed.

Editing a Standalone Broker's Configuration	14
Editing a Broker's Configuration in a Fabric	16

Configuring a broker requires making changes to a number of properties that are stored in multiple files. How the properties are edited depends on how the broker is deployed:

- standalone—if a broker is deployed as a standalone entity and not a part of a fabric, you change the configuration using a combination of editing the broker's configuration file and the console's **config** shell.
- in a fabric—if a broker is deployed into a fabric its configuration is managed by the Fabric Agent which draws all of the configuration from the fabric's registry. To modify the container of a broker running as part of a fabric, you need to modify the profile(s) deployed into it. You can do this by using either the **fabric:profile-edit** console command or Fuse Management Console.

Many of the configuration properties are managed by the OSGi Admin Service and are organized by *persistent identifier* or PID. The container services look in a specific PID for particular properties, so it is important to set the properties in the correct PID.

Editing a Standalone Broker's Configuration

Overview

A standalone broker is one that is not part of a fabric. A standalone broker can, however, be part of a network of broker, a master/slave cluster, or a failover cluster. The distinction is that a standalone is responsible for managing and storing its own configuration.

All of the configuration changes are made directly on the local instance. You make changes using a combination of edits to local configuration files and commands from the console's **config** shell. The configuration that controls the broker's messaging behavior must be edited using an external editor. The configuration the control's the behavior of the broker's runtime container is changed using the console commands.

Editing the messaging configuration

The main broker configuration file, `etc/activemq.xml`, controls the broker's messaging behavior. This includes things like the persistence store in use, any network connectors in use, flow control, and other messaging specific behaviors.

The broker's messaging configuration is specified using Spring XML. It can be edited using any text or XML editor.

Before the configuration changes can take effect, the broker needs to be restarted.

Editing the runtime configuration

Aspects of the broker's runtime behavior such as logging levels, the Web console behavior, and the JMX behavior are controlled by the broker's runtime configuration. These properties are managed by the OSGi Admin Service. You use the command console's **config** shell to edit these properties. The **config** shell has a series of commands that edit properties managed by the OSGi Admin Service:

- **config:list**—lists all of the runtime configuration files and the current values for their properties
- **config:edit**—opens an editing session for a configuration file
- **config:propset**—changes the value of a configuration property
- **config:propdel**—deletes a configuration property
- **config:update**—saves the changes to the configuration file being edited

The changes made using the console take effect as soon as the **config:update** is executed.

For more information about the **config** shell commands, see "[Config Console Commands](#)" in *Console Reference*.



Note

You can edit the property files manually edit the configuration files if you like. They are stored in the `etc/` folder and are named `PID.cfg`.

Editing a Broker's Configuration in a Fabric

Overview

When a broker is part of a fabric, it does not manage its configuration. The broker's configuration is managed by the Fabric Agent. The agent runs along with the broker and updates the broker's configuration based on information from the fabric's registry.

Because the configuration is managed by the Fabric Agent, any changes to the broker's configuration needs to be done by updating the fabric's registry. In a fabric, broker configuration is determined by one or more profiles that are deployed into the broker. To change a broker's configuration, you must update the profile(s) deployed into the broker using either the console's **fabric:** shell or Fuse Management Console.

Profiles

All configuration in a fabric is stored as *profiles* in the Fabric Registry. One or more profiles are assigned to brokers that are part of the fabric. A profile is a collection of configuration that specifies:

- the Apache Karaf features to be deployed
- OSGi bundles to be deployed
- the repositories from which artifacts can be provisioned
- properties that configure the broker's runtime behavior

The configuration profiles are collected into versions. Versions are typically used to make updates to an existing profile without effecting deployed brokers. When a container is configured it is assigned a profile version from which it draws the profiles. Therefore, when you create a new version and edit the profiles in the new version, the profiles that are in use are not changed. When you are ready to test the changes, you can roll them out incrementally by moving brokers to a new version one at a time.

When a broker joins a fabric, a Fabric Agent is deployed with the broker and takes control of the broker's configuration. The agent will ask the Fabric Registry what version and profile(s) are assigned to the broker and configure the broker based on the profiles. The agent will download and install of the

specified bundles and features. It will also set all of the specified configuration properties.

Procedure

The recommended approach to configuring brokers in a fabric is:

1. Create an XML configuration template with property placeholders for any values that you will want to specify on a per-broker basis or that you may want to customize for group of brokers.

Property placeholders allow you to set actual values using fabric properties.

2. Create a base profile for all of the brokers in the fabric using the XML configuration template.
3. Create profiles that inherit from the base profile that will be assigned to one or more brokers.
4. Modify the properties in each of the profiles to the desired values for the brokers to which the profile will be assigned.
5. Assign the new profiles to the desired brokers.

You should always create new profiles or a new version of the existing profiles before making configuration changes. Changes to profiles that are assigned to running brokers take effect immediately. Using new profiles, or a new version, allows you make the changes and test them on a subset of your brokers before rolling the changes to the entire fabric.

Creating a configuration template

To create a configuration template:

1. Create a Fuse MQ Enterprise XML configuration file to use as a template for the broker's configuration.



Tip

You can use the `activemq.xml` file in `InstallDir/etc` as a starting point.

2. Add the XML fragment in [Example 2.1 on page 18](#) to the new XML configuration template.

Example 2.1. Adding Property Placeholder Support to Fuse MQ Enterprise Configuration

```

<!-- Allows us to use system properties and fabric as
variables in this configuration file -->
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="properties">
        <bean class="org.fusesource.mq.fabric.ConfigurationProperties"/>
    </property>
</bean>

<broker ... >
    ...
</broker>

```

This bean add support for using property placeholders in the configuration template.

3. Modify the new XML configuration template to include the desired configuration.

It is recommended that you use property place holders for as many values as possible. As shown in [Example 2.2 on page 18](#), property placeholders are specified using the syntax `${propName}` and can be used as a value for any XML attribute.

Example 2.2. Configuraiton with Property Placeholders

```

<broker xmlns="http://activemq.apache.org/schema/core"
    brokerName="${broker-name}"
    dataDirectory="${data}"
    persistent="${persists}"
    start="false">
    ...
    <persistenceAdapter>
        <jdbcPersistenceAdapter dataDirectory="${data}/derby"
                                dataSource="#derby-ds" />
    </persistenceAdapter>
</broker>

```

The configuration template shown in [Example 2.2 on page 18](#) uses three property placeholders that allow you to modify the base configuration using fabric properties.

Creating a base profile

To create a base profile:

1. Optionally create a new profile version using the **fabric:version-create** command.

This will create a new copy of the existing profiles. See ["fabric:version-create"](#) in *Console Reference* for more information.

2. Import the new XML template into the registry using the **fabric:import** command as shown in [Example 2.4 on page 19](#).

Example 2.3. Importing an XML Configuration Template

```
FuseMQ:karaf@root> fabric:import -t /fabric/configs/versions/version/profiles/mq-base/xmlTemplate xmlTemplatePath
```

See [fabric:import](#) in *Console Reference* for more information.

3. Create a new configuration profile instance to hold the new XML template using the **fabric:mq-create** command as shown in [Example 2.4 on page 19](#).

Example 2.4. Creating a Profile Using an XML Configuration Template

```
FuseMQ:karaf@root> fabric:mq-create --config xmlTemplate profileName
```

This will create a new profile that is based on the default broker profile but uses the imported XML template. See ["fabric:mq-create"](#) in *Console Reference* for more information.

Creating deployment profiles and assigning them to brokers

To create deployment profiles and assigned them to the brokers:

1. Create new profile using the **fabric:profile-create** command as shown in [Example 2.5 on page 19](#).

Example 2.5. Creating a Deployment Profile

```
FuseMQ:karaf@root> fabric:profile-create --parents baseProfile profileName
```

This will create a new profile that inherits from the base profile. See ["fabric:profile-create"](#) in *Console Reference* for more information.

2. Add values for the property placeholders using the **fabric:profile-edit** command as shown in [Example 2.6 on page 20](#).

Example 2.6. Setting Properties in a Profile

```
FuseMQ:karaf@root> fabric:profile-edit -p org.fusesource.mq.fabric.server-profileName/propName=propVal  
profileName
```

The fabric properties for a broker are specified using the PID `org.fusesource.mq.fabric.server-profileName`, so to specify a value for the `broker-name` property for the profile called `myBroker` you would use the command shown in [Example 2.7 on page 20](#).

Example 2.7. Setting the Broker Name Property

```
FuseMQ:karaf@root> fabric:profile-edit -p org.fusesource.mq.fabric.server-myBroker/broker-name=esmeralda  
myBroker
```

3. Assign the new profile to one or more brokers using the **fabric:container-change-profile** command as shown in [Example 2.8 on page 20](#).

Example 2.8. Assigning a Profile to a Broker

```
FuseMQ:karaf@root> fabric:container-change-profile brokerName profileName
```

See ["fabric:container-change-profile"](#) in *Console Reference* for more information.

Using Fuse Management Console

Fuse Management Console simplifies the process of configuring brokers in a fabric by providing an easy to use Web-based interface and reducing the number of steps required to make the changes. For more information on using Fuse Management Console see the [Fuse Management Console Documentation](#)¹.

¹ <http://fusesource.com/documentation/fuse-management-console-documentation>

Chapter 3. Setting up and Accessing the Fuse MQ Enterprise Web Console

Fuse MQ Enterprise's Web console is an easy way to query the broker for JMX statistics, destination statuses, and information on any routes deployed in the broker.

Using the Embedded Console	22
Deploying a Standalone Console	25

The Fuse MQ Enterprise Web console is an embedded console for administering your broker. Using the embedded console is an easy way to manage your broker with minimal configuration.

If you want more security, more reliability, or the ability to monitor master/slave clusters, you can deploy the Web console as a standalone application. It is easy to deploy into Tomcat or any standard Web container.

Using the Embedded Console

Overview

For a standalone broker the default configuration is for the Web console to be loaded along with the broker. The default address for the standalone broker's console is `localhost:8181/activemqweb`.

When a broker is deployed into a fabric, the default configuration profile does not include the Web console. To enable the console, the default configuration profile must be modified to include the feature that implements the Web console. Once it is installed the broker's Web console uses the same default address as a standalone broker.

The default port and security configuration can easily be modified.

Adding the Web console to a broker in a fabric

The default configuration profile for a fabric broker does not include the Fuse MQ Enterprise Web console. To add the Web console to a broker's configuration profile using a broker's console:

1. Create a new configuration profile based on the default `mq` profile using the **`fabric:profile-create`** command as shown in [Example 3.1 on page 22](#).

Example 3.1. Creating a New Profile

```
FuseMQ:karaf@root fabric:profile-create --parent mq mq-console
```

The **`--parent mq`** argument specifies that the `mq` configuration profile is the parent of the new profile. The new profile inherits all of the configuration in the `mq` profile, so initially it is identical to the default.

You can change **`mq-console`** to any name that fits your naming scheme.

For more information on **`fabric:profile-create`** see ["fabric:profile-create"](#) in *Console Reference*.

2. Edit the new profile to include the Web console feature using the **`fabric:profile-edit`** command as shown in [Example 3.2 on page 22](#).

Example 3.2. Editing a Profile to Include the Web Console

```
FuseMQ:karaf@root fabric:profile-edit -f mq-web-console mq-console
```

The `-f mq-web-console` adds the `mq-web-console` feature, which implements the Fuse MQ Enterprise Web console, to the default broker configuration profile.

For more information on **fabric:profile-edit** see ["fabric:profile-edit"](#) in *Console Reference*.

3. Add the new profile to broker you want to monitor with the Web console using the **fabric:container-change-profile** command as shown in [Example 3.3 on page 23](#).

Example 3.3. Adding the Web Console Profile to a Broker

```
FuseMQ:karaf@root fabric:container-change-profile broker1 mq-console
```

This command deploys the configuration profile with the Web console to the broker. The broker's fabric agent will immediately download the required bundles for the Web console and install them.

For more information on **fabric:container-change-profile** see ["fabric:container-change-profile"](#) in *Console Reference*.



Tip

You can also use Fuse Management Console to modify the broker's configuration profile to include the Web console. See the [Fuse Management Console Documentation](#)¹.

Accessing the console

The Fuse MQ Enterprise Web console's address is

`http://hostName:portNum/activemqweb`.

For example, to access the default broker administration console on your local machine, you would point your Web browser at

`http://localhost:8181/activemqweb`.

Changing the port

The port number at which the Web console is accessed is controlled by the broker's Pax Web server. To change the port, you need to edit the `org.osgi.service.http.port` property in the `org.ops4j.pax.web` configuration file.

¹ <http://fusesource.com/documentation/fuse-management-console-documentation>

[Example 3.4 on page 24](#) shows the console commands used to change the port number of a standalone broker's Web console to 8536.

Example 3.4. Changing the Web Console's Port

```
FuseMQ:karaf@root> config:edit org.ops4j.pax.web
FuseMQ:karaf@root> config:proplist
  service.pid = org.ops4j.pax.web
  javax.servlet.context.tempdir = /Applications/FuseMQEnterprise-7.0.0/fuse-mq-7.0.0.fuse-beta-
042/data/pax-web-jsp
  org.osgi.service.http.port = 8181
  org.ops4j.pax.web.config.file = /Applications/FuseMQEnterprise-7.0.0/fuse-mq-7.0.0.fuse-beta-
042/etc/jetty.xml
  org.apache.karaf.features.configKey = org.ops4j.pax.web
FuseMQ:karaf@root> config:propset org.osgi.service.http.port 8536
FuseMQ:karaf@root> config:update
```

[Example 3.5 on page 24](#) shows the console command used to make the same to the Web console of a fabric broker.

Example 3.5. Changing the Web Console's Port in a Fabric

```
FuseMQ:karaf@root> profile-edit -p org.ops4j.pax.web/org.osgi.service.http.port=8536 mq-console
```

Securing the console

The security for the Web console is provided by the Web container in which it is deployed. For the embedded instance of the Web console, you need to configure the embedded Jetty container's security by editing `conf/jetty.xml`.

See ["Web Console Security"](#) in *Security Guide* for details.

Deploying a Standalone Console

Overview

For more security or reliability reasons you can deploy the Web console as a standalone application in Tomcat or another Web container. When running as a standalone application, the Web console can be set up to monitor Master/Slave clusters.

Disabling the embedded console

There is no need to disable the embedded console when using a standalone console. However, If you are using a standalone Web console, there is no reason to use the resources required by the embedded console. Nor is there a reason to leave an extra administrative access point open.

To disable the embedded Web console, you simply need to comment out, or remove, the `import` element that imports the Jetty configuration into your broker's configuration file as shown in [Example 3.6 on page 25](#).

Example 3.6. Disabling the Embedded Web Console

```
<beans ... >

  <broker ... >
    ...
  </broker>

  <!-- <import resource="jetty.xml"/> -->

</beans>
```

Configuring Tomcat

To deploy the Web console in Tomcat 5.x:

1. Download the Web console's WAR, `activemq-web-console-7.0.0.fuse-037.war`, from <http://repo.fusesource.com/nexus/content/groups/m2-release-proxy/org/apache/activemq/activemq-web-console/7.0.0.fuse-037/>.
2. Copy the Web console's WAR to the `TOMCAT_HOME/webapps` folder.
3. Download `activemq-all-7.0.0.fuse-037.jar` from <http://repo.fusesource.com/nexus/content/groups/m2-release-proxy/org/apache/activemq/activemq-all/7.0.0.fuse-037/>.

4. Copy `activemq-all-7.0.0.fuse-037.jar` to the `TOMCAT_HOME/common/lib` folder.
5. Modify `TOMCAT_HOME/bin/catalina.sh(.bat)` to include the configuration in [Example 3.7 on page 26](#).

Example 3.7. Configuration for Deploying the Web Console in Tomcat

```
JAVA_OPTS="-Dwebconsole.type=properties \  
-Dwebconsole.jms.url=brokerURL \  
-Dwebconsole.jmx.url=brokerJMXURL \  
-Dwebconsole.jmx.user=JMXUserName \  
-Dwebconsole.jmx.password=JMXPassword"
```

6. Restart Tomcat.

The Web console will be available at

`tomcatURI/activemq-web-console-7.0.0.fuse-037`.

Monitoring clusters

It's possible to configure the Web console to monitor a master/slave cluster. To do so:

- Specify the JMS URL, `webconsole.jms.url`, with a failover: URI specifying the brokers in the cluster.
- Specify the JMX URL, `webconsole.jmx.url` as a comma separated list that contains the JMX URL for each of the brokers in the cluster.

[Example 3.8 on page 26](#) shows the properties for monitoring a cluster using the Web console.

Example 3.8. Configuration for Monitoring a Cluster with the Web Console

```
-Dwebconsole.jms.url=failover:(tcp://serverA:61616,tcp://serverB:61616)  
-Dwebconsole.jmx.url=service:jmx:rmi:///jndi/rmi://serverA:1099/jmxrmi,ser  
vice:jmx:rmi:///jndi/rmi://serverB:1099/jmxrmi
```

For more information about master/slave clusters see ["Master/Slave"](#) in *Fault Tolerant Messaging*.

Chapter 4. Installing Fuse MQ Enterprise as a Service

Fuse MQ Enterprise can generate a service wrapper that simplifies the process of installing it as a service.

Generating the Service Wrapper	28
Configuring the Wrapper	32
Installing and Starting the Service	36

Installing Fuse MQ Enterprise as a system service is a three step process:

1. [Generate on page 28](#) the service wrapper.
2. [Configure on page 32](#) the service wrapper for your system.
3. [Install on page 36](#) the service wrapper as system service.

Generating the Service Wrapper

Overview

The Fuse MQ Enterprise console's `wrapper` feature generates a wrapper around the Fuse MQ Enterprise runtime that allows you to install a message broker as a system service. The `wrapper` feature does not come preinstalled in the console, so before you can generate the service wrapper you must install the `wrapper` feature.

Once the feature is installed the console gains a **`wrapper:install`** command. Running this command generates a generic service wrapper in the Fuse MQ Enterprise installation.

Installing the wrapper feature

Fuse MQ Enterprise's command shell allows you to provision new functionality using Apache Karaf's feature mechanism. One of the features that can be installed this way is the one that creates the wrapper used to install Fuse MQ Enterprise as a system service.

To install the wrapper feature:

1. Start Fuse MQ Enterprise in console mode using the **`fusemq`** command.
2. Once the console is started and the command prompt appears, enter **`features:install wrapper`**.

The **`features:install`** command will locate the required libraries to provision the wrapper feature and deploy it into the run time. For more information see ["features:install"](#) in *Console Reference*.

3. Verify that the feature was installed by entering **`wrapper:install --help`**.

You should see the output shown in [Example 4.1 on page 28](#).

Example 4.1. Wrapper Install Help

```
DESCRIPTION
    wrapper:install

    Install the container as a system service in the OS.

SYNTAX
    wrapper:install [options]
```

```
OPTIONS
--help
    Display this help message
-d, --display
    The display name of the service.
-s, --start-type
    Mode in which the service is installed.
    AUTO_START or
    DEMAND_START (Default: AUTO_START)
    (defaults to AUTO_START)
-n, --name
    The service name that will be used when in
    stalling the service.
    (Default: karaf)
    (defaults to karaf)
-D, --description
    The description of the service.
    (defaults to )
```

Using the wrapper feature

The wrapper feature adds a single command to the shell: **wrapper:install**. This command generates the files that make up the service wrapper. The options described in [Table 4.1 on page 29](#) allow you to customize some features of the service created using the wrapper.

Table 4.1. Service Wrapper Installation Options

Options	Description
-d, --display	Specifies the name displayed for the service by the OS.
-s, --start-type	Specifies the node in which the service is installed. Can be either AUTO_START or DEMAND_START. The default is AUTO_START.
-n, --name	Specifies the service name that will be used when installing the service. The default value is karaf.
-D, --description	Specifies a description of the service.

For example, you would use the **wrapper:install** command shown in [Example 4.2 on page 30](#) to create a service named FuseMQ that would start automatically start when the machine is booted.

Example 4.2. Generating a Service Wrapper

```
FuseMQ:karaf@root> wrapper:install -n FuseMQ -d FuseMQ -D "FuseMQ Broker"
Creating file: InstallDir\bin\FuseMQ-wrapper.exe
Creating file: InstallDir\bin\FuseMQ-service.bat
Creating file: InstallDir\etc\FuseMQ-wrapper.conf
Creating file: InstallDir\lib\libwrapper.so
Creating file: InstallDir\lib\karaf-wrapper.jar
Creating file: InstallDir\lib\karaf-wrapper-main.jar

Setup complete. You may wish to tweak the JVM properties in the wrapper configuration file:
    InstallDir\etc\FuseMQ-wrapper.conf
before installing and starting the service.

To install the service, run:
    C:> InstallDir\bin\FuseMQ-service.bat install

Once installed, to start the service run:
    C:> net start "FuseMQ"

Once running, to stop the service run:
    C:> net stop "FuseMQ"

Once stopped, to remove the installed the service run:
    C:> InstallDir\bin\karaf-service.bat remove
```

Generated files

The following files are generated and make up the service wrapper:

- etc\ServiceName-wrapper(.exe)—the executable file for the wrapper.
- bin\ServiceName-service(.bat)—the script used to install and remove the service.
- etc\ServiceName-wrapper.conf—the wrapper's configuration file.
- Three library files required by the service wrapper:
 - lib\libwrapper.so
 - lib\karaf-wrapper.jar
 - lib\karaf-wrapper-main.jar



Important

The only generated file you should modify is the configuration file.

Configuring the Wrapper

Overview

The service wrapper is configured by the `ServiceName-wrapper.conf` file, which is located under the `InstallDir/etc/` directory.

There are several settings you may want to change including:

- the default environment settings
- the properties passed to the JVM
- the classpath
- the JMX settings
- the logging settings

Specifying the Fuse MQ Enterprise's environment

A broker's environment is controlled by three environment variables:

- `KARAF_HOME`—the location of the Fuse MQ Enterprise install directory.
- `KARAF_BASE`—the root directory containing the configuration and OSGi data specific to the broker instance.

The configuration for the broker instance is stored in the `KARAF_BASE/conf` directory. Other data relating to the OSGi runtime is also stored beneath the base directory.

- `KARAF_DATA`—the directory containing the logging and persistence data for the broker.

[Example 4.3 on page 33](#) shows the default values.

Example 4.3. Default Environment Settings

```
set.default.KARAF_HOME=InstallDir
set.default.KARAF_BASE=InstallDir
set.default.KARAF_DATA=InstallDir\data
```

Passing parameters to the JVM

If you want to pass parameters to the JVM, you do so by setting wrapper properties using the form `wrapper.java.additional.<n>`. `<n>` is a sequence number that must be distinct for each parameter.

One of the most useful things you can do by passing additional parameters to the JVM is to set Java system properties. The syntax for setting a Java system property is `wrapper.java.additional.<n>=-DPropName=PropValue`.

[Example 4.4 on page 33](#) shows the default Java properties.

Example 4.4. Default Java System Properties

```
# JVM
# note that n is the parameter number starting from 1.
wrapper.java.additional.1=-Dkaraf.home="%KARAF_HOME%"
wrapper.java.additional.2=-Dkaraf.base="%KARAF_BASE%"
wrapper.java.additional.3=-Dkaraf.data="%KARAF_DATA%"
wrapper.java.additional.4=-Dcom.sun.management.jmxremote
wrapper.java.additional.5=-Dkaraf.startLocalConsole=false
wrapper.java.additional.6=-Dkaraf.startRemoteShell=true
wrapper.java.additional.7=-Djava.endorsed.dirs="%JAVA_HOME%/jre/lib/en
dorsed;%JAVA_HOME%/lib/endorsed;%KARAF_HOME%/lib/endorsed"
wrapper.java.additional.8=-
Djava.ext.dirs="%JAVA_HOME%/jre/lib/ext;%JAVA_HOME%/lib/ext;%KARAF_HOME%/lib/ext"
```

Adding classpath entries

You add classpath entries using the syntax `wrapper.java.classpath.<n>`. `<n>` is a sequence number that must be distinct for each classpath entry.

[Example 4.5 on page 33](#) shows the default classpath entries.

Example 4.5. Default Wrapper Classpath

```
wrapper.java.classpath.1=%KARAF_BASE%/lib/karaf-wrapper.jar
wrapper.java.classpath.2=%KARAF_HOME%/lib/karaf.jar
```

```
wrapper.java.classpath.3=%KARAF_HOME%/lib/karaf-jaas-boot.jar
wrapper.java.classpath.4=%KARAF_BASE%/lib/karaf-wrapper-main.jar
```

JMX configuration

The default service wrapper configuration does not enable JMX. It does, however, include template properties for enabling JMX. To enable JMX:

1. Locate the line `# Uncomment to enable jmx.`

There are three properties, shown in [Example 4.6 on page 34](#), that are used to configure JMX.

Example 4.6. Wrapper JMX Properties

```
# Uncomment to enable jmx
#wrapper.java.additional.n=-Dcom.sun.management.jmxre
mote.port=1616
#wrapper.java.additional.n=-Dcom.sun.management.jmxre
mote.authenticate=false
#wrapper.java.additional.n=-Dcom.sun.management.jmxre
mote.ssl=false
```

2. Remove the `#` from in front of each of the properties.
3. Replace the `n` in each property to a number that fits into the sequence of addition properties established in the configuration.

You can change the settings to use a different port or secure the JMX connection.

For more information about using JMX see ["Using JMX" on page 59](#).

Configuring logging

The wrapper's logging is configured using the properties described in [Table 4.2 on page 34](#).

Table 4.2. Wrapper Logging Properties

Property	Description
<code>wrapper.console.format</code>	Specifies how the logging information sent to the console is formatted. The format consists of the following tokens: <ul style="list-style-type: none"> • <code>L</code>—log level

Property	Description
	<ul style="list-style-type: none"> • P—prefix • D—thread name • T—time • Z—time in milliseconds • U—approximate uptime in seconds (based on internal tick counter) • M—message
<code>wrapper.console.loglevel</code>	Specifies the logging level displayed on the console.
<code>wrapper.logfile</code>	Specifies the file used to store the log.
<code>wrapper.logfile.format</code>	Specifies how the logging information sent to the log file is formatted.
<code>wrapper.console.loglevel</code>	Specifies the logging level sent to the log file.
<code>wrapper.console.maxsize</code>	Specifies the maximum size, in bytes, that the log file can grow to before the log is archived. The default value of 0 disables log rolling.
<code>wrapper.console.maxfiles</code>	Specifies the maximum number of archived log files which will be allowed before old files are deleted. The default value of 0 implies no limit.
<code>wrapper.syslog.loglevel</code>	Specifies the logging level for the sys/event log output.

For more information about Fuse MQ Enterprise logging see ["Using Logging" on page 53](#).

Installing and Starting the Service

Overview

The operating system determines the exact steps using to complete the installation of Fuse MQ Enterprise as a service. The **wrapper:install** command provides basic instructions for your operating system.

Windows

To install the service run `InstallDir\bin\ServiceName-service.bat install`. If you used the default start setting, the service will start when Windows is launched. If you specified `DEMAND_START`, you will need to start the service manually.

To start the service manually run `net start "ServiceName"`. You can also use the Windows service UI.

To manually stop the service run `net stop "ServiceName"` You can also use the Windows service UI.

You remove the installed the service by running `InstallDir\bin\ServiceName-service.bat remove`.

Redhat Linux

To install the service and configure it to start when the machine boots, run the following commands:

```
# ln -s InstallDir\bin\ServiceName-service /etc/init.d/  
# chkconfig ServiceName-service --add  
# chkconfig ServiceName-service on
```

To start the service manually run `service ServiceName-service start`.

To manually stop the service run `service ServiceName-service stop`.

You remove the installed the service by running the following commands:

```
#service ServiceName-service stop  
# chkconfig ServiceName-service --del  
# rm /etc/init.d/ServiceName-service
```

Ubuntu Linux

To install the service and configure it to start when the machine boots, run the following commands:

```
# ln -s InstallDir\bin\ServiceName-service /etc/init.d/  
# update-rc.d ServiceName-service defaults
```

To start the service manually run `/etc/init.d/ServiceName-service start`.

To manually stop the service run `/etc/init.d/ServiceName-service stop`.

You remove the installed the service by running the following commands:

```
#/etc/init.d/ServiceName-service stop  
# rm /etc/init.d/ServiceName-service
```


Chapter 5. Starting a Broker

You start a broker using a simple command. The broker can either be started so that it launches a command console or so that it runs as a daemon process. When a broker is part of a fabric, you can remotely start the broker remotely.

Overview

A broker can be run in one of two modes:

- console mode—the broker starts up as a foreground process and presents the user with a command shell
- daemon mode—the broker starts up as a background process that can be managed using a remote console or the provided command line tools

The default location for the broker's configuration for the broker is the `InstallDir/etc/activemq.xml` configuration file. The configuration uses values loaded from the `InstallDir/etc/system.properties` file and the `InstallDir/etc/org.fusesource.mq.fabric.server-default.cfg` file.

Starting in console mode

When you start the broker in console mode you will be placed into a command shell that provides access to a number of commands for managing the broker and its OSGi runtime.



Important

When the broker is started in console mode, you cannot close the console without killing the broker.

To launch a broker in console mode, change to `InstallDir` and run one of the commands in [Table 5.1 on page 39](#).

Table 5.1. Start up Commands for Console Mode

Windows	<code>bin\fusemq.bat</code>
Unix	<code>bin/fusemq</code>

If the server starts up correctly you should see something similar to [Example 5.1 on page 40](#) on the console.

Example 5.1. Broker Console

[illegible]

Starting in daemon mode

Launching a broker in daemon mode runs Fuse MQ Enterprise in the background without a console. To launch a broker in daemon mode, change to *InstallDir* and run one of the commands in [Table 5.2 on page 40](#).

Table 5.2. Start up Commands for Daemon Mode

Windows	bin\start.bat
Unix	bin/start

Starting a broker in a fabric

If a broker is deployed as part of a fabric you can start it remotely in one of three ways:

- using the console of one of the other broker's in the fabric

If one of the brokers in the fabric is running in console mode you can use the **fabric:container-start** command to start any of the other brokers in the fabric. The command requires that you supply the container name used when creating the broker in the fabric. For example to start a broker named `fabric-broker3` you would use the command shown in [Example 5.2 on page 40](#).

Example 5.2. Starting a Broker in a Fabric

```
FuseMQ:karaf@root> fabric:container-start fabric-broker3
```

For more information see ["fabric:container-start"](#) in *Console Reference*.

- using the administration client of one of the broker's in the fabric

If none of the brokers are running in console mode, you can use the administration client on one of the brokers to execute the **fabric:container-start** command. The administration client is run using the **client** command in Fuse MQ Enterprise's `bin` folder.

[Example 5.3 on page 41](#) shows how to use the remote client to start remote broker in the fabric.

Example 5.3. Starting a Broker in a Fabric with the Administration Client

```
bin/client fabric:container-start fabric-broker3
```

- using Fuse Management Console

Fuse Management Console can start and stop any of the brokers in the fabric it manages from a Web based console.

For more information see the [Fuse Management Console Documentation](#)¹.

¹ <http://fusesource.com/docs/fmc>

Chapter 6. Sending Commands to the Broker

Fuse MQ Enterprise provides a number of commands that can be used to manage a broker, deploy new brokers, and report administrative details. You can send these commands to a broker using either the broker command console or the administration client.

Overview

The default mode for running a Fuse MQ Enterprise broker is to run in daemon mode. In this mode, the broker runs as a background process and you have no direct means for managing it or requesting status information. You can access a broker in daemon mode in the following ways:

- the Fuse MQ Enterprise administration client that can be used to send any of the console commands to a broker running in daemon mode
- a broker running in console mode can connect to a remote broker and be used to manage the remote broker
- Fuse MQ Enterprise includes a vanilla Apache Karaf shell that can connect to a remote broker and be used to manage the remote broker

If a broker is started in console mode, you can simply enter commands directly in the command console.

Running the administration client

The Fuse MQ Enterprise administration client is run using the **client** in `InstallDir/bin`. [Example 6.1 on page 43](#) shows the syntax for the command.

Example 6.1. Client Command

```
client [--help] [-a port] [-h host] [-u user] [-p password] [-v] [-r attempts] [-d delay] [commands]
```

[Table 6.1 on page 43](#) describes the command's arguments.

Table 6.1. Administration Client Arguments

Argument	Description
--help	Displays the help message.

Argument	Description
-a	Specifies the remote host's port.
-h	Specify the remote host's name.
-u	Specifies user name used to log into the broker.
-p	Specifies the password used to log into the broker.
-v	Use verbose output.
-r	Specifies the maximum number of attempts to establish a connection.
-d	Specifies, in seconds, the delay between retries. The default is 2 seconds.
<i>commands</i>	Specifies one or more commands to run. If no commands are specified, the client enters an interactive mode.

Using the broker console

The console provides commands that you can use to perform basic management of your Fuse MQ Enterprise environment, including managing destinations, connections and other administrative objects in the broker.

The console uses prefixes to group commands relating to the same functionality. For example commands related to configuration are prefixed **config:**, and logging-related commands are prefixed **log:**.

The console provides two levels of help:

- console help—list all of the commands along with a brief summary of the commands function
- command help—a detailed description of a command and its arguments

To access the console help you use the **help** command from the console prompt. It will display a grouped list of all the commands available in the console. Each command in the list will be followed by a description of the command as shown in [Example 6.2 on page 44](#).

Example 6.2. Console Help

```
FuseMQ:karaf@root> help
COMMANDS
    activemq:browse
    activemq:bstat
```

```

    activemq:create-broker
        Creates a broker instance.
    activemq:destroy-broker
        Destory a broker instance.
    activemq:list
    activemq:purge
    activemq:query
    admin:change-opts
        Changes the Java options of an existing container
instance.
    admin:change-rmi-registry-port
        Changes the RMI registry port (used by management
layer) of an
        existing container instance.
    ...
FuseMQ:karaf@root>

```

The help for each command includes the definition, the syntax, and the arguments and any options. To display the help for a command, type the command with the `--help` option. As shown in [Example 6.3 on page 45](#), entering `admin:start --help` displays the help for that command.

Example 6.3. Help for a Command

```

FuseMQ:karaf@root> admin:start --help
DESCRIPTION
    admin:start

    Starts an existing container instance.

SYNTAX
    admin:start [options] name

ARGUMENTS
    name                The name of the container instance

OPTIONS
    --help
                        Display this help message
    -o, --java-opts
                        Java options when launching the instance

FuseMQ:karaf@root>

```

Connecting a console to a remote broker

How you connect a command console to a broker on a remote machine depends on if the brokers are part of the same fabric. If the remote broker you want to command is a part of the same fabric as the broker whose

command console you are using, then you can use the **fabric:container-connect** command to establish a connection to the remote broker.

The **fabric:container-connect** command has one required argument that specifies the name of the container to which a connection will be opened. You can also specify a command to be executed by the remote console connection. If you do not specify a command, you are presented with a prompt that will pass commands to the remote broker's console. For more details about **fabric:container-connect**, see ["fabric:container-connect"](#) in *Console Reference*.

If you are not using fabric, or the remote broker is not part of the same fabric as the broker whose command console you are using, you create a remote connection using the **ssh:ssh** command. The **ssh:ssh** command also only requires a single argument to establish the remote connection. In this case, it is the hostname, or IP address, of the machine on which the broker is running. If the remote broker is not using the default SSH port (8101), you will also need to specify the remote broker's SSH port using the `-p` flag. You can also specify a command to be executed by the remote console connection. If you do not specify a command, you are presented with a prompt that will pass commands to the remote broker's console. For more details about **ssh:ssh**, see [ssh:ssh](#), in *Console Reference*.

To disconnect from the remote console, you use the **logout** command or press Control+D.

Starting a basic console

Available commands

The remote client can execute any of the broker's console commands. For a complete list of commands see the [Console Reference](#).

Chapter 7. Shutting Down a Broker

Brokers can be shutdown from either the machine on which they are running or remotely from a different machine.

Shutting Down a Local Broker	48
Shutting Down a Broker Remotely	49



Important

If the broker is running in console mode it can only be shutdown locally.

Shutting Down a Local Broker

Overview

The method used to stop a broker running on the machine you logged into depends on the mode in which the broker is running. If it is running in console mode, you use one of the console commands to shut down the broker. If it is running in daemon mode, the broker doesn't have a command console. So, you need to use one of the utility commands supplied with Fuse MQ Enterprise.

Stopping the broker from console mode

If you launched Fuse MQ Enterprise by running **fusemq**, you shut it down using the **shutdown -f** command as shown in [Example 7.1 on page 48](#).

Example 7.1. Using the Console's Shutdown Command

```
FuseMQ:karaf@root> shutdown -f
FuseMQ:karaf@root>
logout

[Process completed]
```



Tip

CTRL+D will also shutdown the broker.

Stopping a broker running in daemon mode

If you launched Fuse MQ Enterprise by running the **start** command, log in to the machine where the broker is running and run the **stop** command in the broker installation's `bin` folder.



Tip

You can stop a broker running in daemon mode remotely. See ["Shutting Down a Broker Remotely" on page 49](#).

Shutting Down a Broker Remotely

Overview

For many use cases logging into the machine running a broker instance is impractical. In those cases, you need a way to stop a broker from a remote machine. Fuse MQ Enterprise offers a number of ways to accomplish this task:

- using the **stop** command—the stop command does not require starting an instance of the broker
- using a remote console connection—a broker's console can be used to remotely shutdown a broker on another machine
- using a fabric member's console—brokers that are part of a fabric can stop members of their fabric
- using Fuse Management Console—brokers that are part of a fabric can be stopped using an Fuse Management Console connected to their fabric

For more information see the [Fuse Management Console Documentation](#)¹.

Using the stop command

You can stop a remote instance without starting up Fuse ESB Enterprise on your local host by running the **stop** command in the `InstallDir/bin` directory. The commands syntax is shown in [Example 7.2 on page 49](#).

Example 7.2. Stop Command Syntax

```
stop [-a port] {-h hostname} {-u username} {-p password}
```

`-a port`

Specifies the SSH port of the remote instance. The default is 8101.

`-h hostname`

Specifies the hostname of the machine on which the remote instance is running.

`-u username`

Specifies the username used to connect to the remote broker.

¹ <http://fusesource.com/docs/fmc>



Tip

The default username for a broker is `karaf`.

```
-p password
```

Specifies the password used to connect to the remote broker.



Tip

The default password for a broker is `karaf`.

[Example 7.3 on page 50](#) shows how to stop a remote broker on a machine named `NEBrokerHost2`.

Example 7.3. Stopping a Remote Broker

```
bin/stop -u karaf -p karaf -h NEBrokerHost2
```

Using a remote console

Fuse MQ Enterprise's console can be connected to a remote broker using the **ssh:ssh** command. Once the console is connected to the remote broker, you can shut it down by running the **osgi:shutdown** command.

[Example 7.4 on page 50](#) shows the command sequence for using a remote console connection to shutdown a broker running on a machine named `NWBrokerHost`.

Example 7.4. Shutting Down a Broker using a Remote Console Connection

```
FuseMQ:karaf@root> ssh -l karaf -P karaf NWBrokerHost
```

```
|  _  |          |  \|  |  _  | | | | | | |
| |__| |__| |__| |  .  |  |  |
| |__| |__| |__| | /  \ |  |  |
| |__| |__| |__| | \|  \|  |  |
\_|  \_|  \_|  \_|  \|  \|  \_|  \_|
    Fuse MQ (7.0.0.fuse-beta-040)
    http://fusesource.org/mq/
```

```
Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or 'osgi:shutdown' to shutdown Fuse MQ.
```

```
FuseMQ:karaf@root> osgi:shutdown
Confirm: shutdown instance root (yes/no):
yes
FuseMQ:karaf@root> FuseESB:karaf@root>
```



Important

Pressing Control+D when connected to a remote broker closes the remote connection and returns you to the local shell.

Shutting down remote brokers in a fabric

If the broker you want to shutdown is part of a fabric, you can shut it down from any of the brokers in the fabric using the **fabric:container-stop** console command. **fabric:container-stop** takes the name of the fabric container hosting the broker as an argument. The command can be run either from a broker in console mode or using the broker's administration client.

[Example 7.5 on page 51](#) shows how to use the administration client to shutdown a broker running in a container named `fabric-broker3`.

Example 7.5. Shutting Down a Broker in a Fabric

```
./bin/client fabric-broker3
```

For more information see ["fabric:container-stop"](#) in *Console Reference*.

Chapter 8. Using Logging

The broker's log contains information about all of the critical events that occur in the broker. You can configure the granularity of the logged messages to provide the required amount of detail.

Logging Configuration	54
Viewing the Log	56

Fuse MQ Enterprise uses the *OPS4j Pax Logging* system, a standard OSGi logging service that also supports the following APIs:

- Apache Log4j
- Apache Commons Logging
- SLF4J
- Java Util Logging

Logging Configuration

Overview

Fuse MQ Enterprise uses the Pax Logging framework, so configuring a broker's logging behavior involves setting properties in both the Pax Logging configuration and the broker's runtime configuration. The Pax Logging configuration is used to control logging levels and where logs are saved. The Fuse MQ Enterprise runtime logging configuration controls the initial logging level and how log information is displayed in the Fuse MQ Enterprise console.

Configuration files

Three configuration files are used to configuring Fuse MQ Enterprise logging:

- `org.ops4j.pax.logging.cfg`—the main logging configuration file. By default it sets the root logger's level to `INFO` and defines two appenders: one for the console and one for the log file. Logging configuration uses standard Log4j configuration.
- `org.apache.karaf.log.cfg`—configures the output of the **log** console commands.
- `system.properties`—contains one property, `org.ops4j.pax.logging.DefaultServiceLog.level`, which sets the logging level early in the boot process when the pax-logging service is not yet available. This is set to `ERROR` by default.

For standalone brokers these configuration files are found in the `InstallDir/etc` directory.

Changing the log levels

The default logging configuration sets the logging levels so that the log file will provide enough information to monitor the behavior of the runtime and provide clues about what caused a problem. However, the default configuration will not provide enough information to debug most problems.

The most useful logger to change when trying to debug an issue with Fuse MQ Enterprise is the root logger. You will want to set its logging level to `DEBUG` in the `org.ops4j.pax.logging.cfg` file as shown in [Example 8.1 on page 55](#).

Example 8.1. Changing Logging Levels

```
# Root logger
log4j.rootLogger=DEBUG, out, osgi:VmLogAppender
...

```

**Tip**

You can also change a standalone broker's logging level using the **log:set** console command. See [log:set](#), in *Console Reference*.

Changing the appenders' thresholds

When debugging a problem in Fuse MQ Enterprise you may want to change the level of logging information that is displayed on the console.

[Example 8.2 on page 55](#) shows an example of setting the root logger to DEBUG but limiting the information displayed on the console to WARN.

Example 8.2. Changing the Log Information Displayed on the Console

```
log4j.rootLogger=DEBUG, stdout, osgi:VmLogAppender
...

log4j.appender.stdout.threshold=WARN
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE}
| %-5.5p | %-16.16t | %-32.32c{1} | %-32.32C %4L | %m%n
...

```

More information

For more information on Pax logging, see <http://wiki.ops4j.org/display/paxlogging/Pax+Logging>.

Viewing the Log

Overview

There are three ways you can view the log:

- using a text editor
 - using the broker's, or a remote broker's, console
 - using the administration client
-

Viewing the log in a text editor

The log files are stored as simple text files in `InstallDir/data/log`. The main log file is `karaf.log`. If archiving is turned on, there may be archived log files also stored in the logging directory.

Log entries are listed in chronological order with the oldest entries first. The default output displays the following information:

- the time of the entry
 - the log level of the entry
 - the thread that generated the entry
 - the bundle that generated the entry
 - an informational message about the cause of the entry
-

Viewing the log with the console

The Fuse MQ Enterprise console provides the following commands for viewing the log:

- **log:display**—displays the most recent log entries

By default, the number of entries returned and the pattern of the output depends on the `size` and `pattern` properties in the `org.apache.karaf.log.cfg` file. You can override these using the `-p` and `-d` arguments.

- **log:display-exception**—displays the most recently logged exception
- **log:tail**—continuously display log entries

For more information on using these commands see ["Log Console Commands"](#) in *Console Reference*.

Viewing the log with the administration client

If you do not have a broker running in console mode, you can also use the administration client to invoke the broker's log displaying commands. For example, entering `client log:display` into a system terminal will display the most recent log entries for the local broker.

Chapter 9. Using JMX

Fuse MQ Enterprise is fully instrumented to provide statistics about its performance using JMX. You can monitor a broker using any JMX aware monitoring tool.

Configuring JMX	60
Statistics Collected by JMX	62
Managing the Broker with JMX	65

By default Fuse MQ Enterprise creates MBeans, loads them into the MBean server created by the JVM, and creates a dedicated JMX connector that provides a Fuse MQ Enterprise-specific view of the MBean server. The default settings are sufficient for simple deployments and make it easy to access the statistics and management operations provided by a broker. For more complex deployments you easily configure many aspects of how a broker configures itself for access through JMX. For example, you can change the JMX URI of the JMX connector created by the broker or force the broker to use the generic JMX connector created by the JVM.

By connecting a JMX aware management and monitoring tool to a broker's JMX connector, you can view detailed information about the broker. This information provides a good indication of broker health and potential problem areas. In addition to the collected statistics, Fuse MQ Enterprise's JMX interface provides a number of operations that make it easy to manage a broker instance. These include stopping a broker, starting and stopping network connectors, and managing destinations.

Configuring JMX

Overview

By default a broker is set up to allow for JMX management. It uses the JVM's MBean server and creates its own JMX connector at

`service:jmx:rmi:///jndi/rmi://hostname:1099/jmxrmi`. If the default configuration does not meet the needs of the deployment environment, the broker provides configuration properties for customizing most aspects of its JMX behavior. For instance, you can completely disable JMX for a broker. You can also force the broker to create its own MBean server.

Enabling and disabling

By default JMX is enabled for a Fuse MQ Enterprise broker. To disable JMX entirely you simply set the `broker` element's `useJmx` attribute to `false`. This will stop the broker from exposing itself via JMX.



Important

Disabling JMX will also disable the command-line administrative tools.

Securing access to JMX

In a production environment it is advisable to secure the access to your brokers' management interfaces. The steps to secure access to a broker's JMX interface depends on the JMX connector the broker uses.

If the broker creates its own JMX connector you configure the security options directly in the broker's configuration. If the broker uses the JMX connector provided by the JVM, you need to modify the scripts used to start the broker to pass the security configuration to the JVM.

For a detailed description of the steps required see ["JMX Security"](#) in *Security Guide*.

Advanced configuration

If the default JMX behavior is not appropriate for your deployment environment, you can customize how the broker exposes its MBeans. To customize a broker's JMX configuration, you add a `managementContext` child element to the broker's `broker` element. The `managementContext` element uses a `managementContext` child to configure the broker. The attributes of the inner `managementContext` element specify the broker's JMX configuration.

[Table 9.1 on page 61](#) describes the configuration properties for controlling a broker's JMX behavior.

Table 9.1. Broker JMX Configuration Properties

Property	Default Value	Description
<code>useMBeanServer</code>	<code>true</code>	Specifies whether the broker will use the MBean server created by the JVM. When set to <code>false</code> , the broker will create an MBean server.
<code>jmxDomainName</code>	<code>org.apache.activemq</code>	Specifies the JMX domain used by the broker's MBeans.
<code>createMBeanServer</code>	<code>true</code>	Specifies whether the broker creates an MBean server if none is found.
<code>createConnector</code>	<code>true</code>	Specifies whether the broker creates a JMX connector for the MBean server. If this is set to <code>false</code> the broker will only be accessible using the JMX connector created by the JVM.
<code>connectorPort</code>	<code>1099</code>	Specifies the port number used by the JMX connector created by the broker.
<code>connectorHost</code>	<code>localhost</code>	Specifies the host used by the JMX connector and the RMI server.
<code>rmiServerPort</code>	<code>0</code>	Specifies the RMI server port. This setting is useful if port usage needs to be restricted behind a firewall.
<code>connectorPath</code>	<code>/jmxrmi</code>	Specifies the path under which the JMX connector will be registered.

[Example 9.1 on page 61](#) shows configuration for a broker that will only use the JVM's MBean server and will not create its own JMX connector.

Example 9.1. Configuring a Broker's JMX Connection

```
<broker ... >
  ...
  <managementContext>
    <managementContext createMBeanServer="false"
                        createConnector="false" />
  </managementContext>
  ...
</broker>
```

Statistics Collected by JMX

Broker statistics

[Table 9.2 on page 62](#) describes the statistics collected for a broker.

Table 9.2. Broker JMX Statistics

Name	Description
BrokerId	Specifies the broker's unique ID.
BrokerName	Specifies the broker's name.
BrokerVersion	Specifies the version of the broker.
DataDirectory	Specifies the pathname of the broker's data directory.
TotalEnqueueCount	Specifies the total number of messages that have been sent to the broker.
TotalDequeueCount	Specifies the number of messages that have been acknowledged on the broker.
TotalConsumerCount	Specifies the number of message consumers subscribed to destinations on the broker.
TotalProducerCount	Specifies the number of message producers active on destinations on the broker.
TotalMessageCount	Specifies the number of unacknowledged messages on the broker.
MemoryLimit	Specifies the memory limit, in bytes, used for holding undelivered messages before paging to temporary storage.
MemoryPercentageUsed	Specifies the percentage of available memory in use.
StoreLimit	Specifies the disk space limit, in bytes, used for persistent messages before producers are blocked.
StorePercentageUsed	Specifies the percentage of the store space in use.
TempLimit	Specifies the disk space limit, in bytes, used for non-persistent messages and temporary data before producers are blocked.
TempPercentageUsed	Specifies the percentage of available temp space in use.

Destination statistics

[Table 9.3 on page 62](#) describes the statistics collected for a destination.

Table 9.3. Destination JMX Statistics

Name	Description
BlockedProducerWarningInterval	Specifies, in milliseconds, the interval between warnings issued when a producer is blocked from adding messages to the destination.

Name	Description
MemoryLimit	Specifies the memory limit, in bytes, used for holding undelivered messages before paging to temporary storage.
MemoryPercentageUsed	Specifies the percentage of available memory in use.
MaxPageSize	Specifies the maximum number of messages that can be paged into the destination.
CursorFull	Specifies if the cursor has reached its memory limit for paged messages.
CursorMemoryUsage	Specifies, in bytes, the amount of memory the cursor is using.
CursorPercentUsage	Specifies the percentage of the cursor's available memory is in use.
EnqueueCount	Specifies the number of messages that have been sent to the destination.
DequeueCount	Specifies the number of messages that have been acknowledged and removed from the destination.
DispatchCount	Specifies the number of messages that have been delivered to consumers, but not necessarily acknowledged by the consumer.
InFlightCount	Specifies the number of dispatched to, but not acknowledged by, consumers.
ExpiredCount	Specifies the number of messages that have expired in the destination.
ConsumerCount	Specifies the number of consumers that are subscribed to the destination.
QueueSize	Specifies the number of messages in the destination that are waiting to be consumed.
AverageEnqueueTime	Specifies the average amount of time, in milliseconds, that messages sat in the destination before being consumed.
MaxEnqueueTime	Specifies the longest amount of time, in milliseconds, that a message sat in the destination before being consumed.
MinEnqueueTime	Specifies the shortest amount of time, in milliseconds, that a message sat in the destination before being consumed.
MemoryUsagePortion	Specifies the portion of the broker's memory limit used by the destination.
ProducerCount	Specifies the number of producers connected to the destination.

Subscription statistics

[Table 9.4 on page 63](#) describes the statistics collected for a subscription.

Table 9.4. Connection JMX Statistics

Name	Description
EnqueueCounter	Counts the number of messages that matched the subscription.

Name	Description
DequeueCounter	Counts the number of messages were sent to and acknowledge by the client.
DispatchedQueueSize	Specifies the number of messages dispatched to the client and are awaiting acknowledgement.
DispatchedCounter	Counts the number of messages that have been sent to the client.
MessageCountAwaitingAcknowledge	Specifies the number of messages dispatched to the client and are awaiting acknowledgement.
Active	Specifies if the subscription is active.
PendingQueueSize	Specifies the number of messages pending delivery.
PrefetchSize	Specifies the number of messages to pre-fetch and dispatch to the client.
MaximumPendingMessageLimit	Specifies the maximum number of pending messages allowed.

Managing the Broker with JMX

Overview

The MBeans exposed by Fuse MQ Enterprise provide a number of operations for monitoring and managing a broker instance. You can access these operations through any tool that supports JMX.

Broker actions

[Table 9.5 on page 65](#) describes the operations exposed by the MBean for a broker.

Table 9.5. Broker MBean Operations

Operation	Description
<code>void start();</code>	Starts the broker. In reality this operation is not useful because you cannot access the MBeans if the broker is stopped.
<code>void stop();</code>	Forces a broker to shut down. There is no guarantee that all messages will be properly recorded in the persistent store.
<code>void stopGracefully(String queueName);</code>	Checks that all listed queues are empty before shutting down the broker.
<code>void enableStatistics();</code>	Activates the broker's statistics plug-in.
<code>void resetStatistics();</code>	Resets the data collected by the statistics plug-in.
<code>void disableStatistics();</code>	Deactivates the broker's statistics plug-in.
<code>String addConnector(String URI);</code>	Adds a transport connector to the broker and starts it listening for incoming client connections and returns the name of the connector.
<code>boolean removeConnector(String connectorName);</code>	Deactivates the specified transport connector and removes it from the broker.
<code>String addNetworkConnector(String URI);</code>	Adds a network connector to the specified broker and returns the name of the connector.

Operation	Description
<code>boolean removeNetworkConnector(String connectorName);</code>	Deactivates the specified connector and removes it from the broker.
<code>void addTopic(String name);</code>	Adds a topic destination to the broker.
<code>void addQueue(String name);</code>	Adds a queue destination to the broker.
<code>void removeTopic(String name);</code>	Removes the specified topic destination from the broker.
<code>void removeQueue(String name);</code>	Removes the specified queue destination from the broker.
<code>ObjectName createDurableSubscriber(String clientId, String subscriberId, String topicName, String selector);</code>	Creates a new durable subscriber.
<code>void destroyDurableSubscriber(String clientId, String subscriberId);</code>	Destroys a durable subscriber.
<code>void gc();</code>	Runs the JVM garbage cleaner.
<code>void terminateJVM(int exitCode);</code>	Shuts down the JVM.
<code>void reloadLog4jProperties();</code>	Reloads the logging configuration from <code>log4j.properties</code> .

Connector actions

[Table 9.6 on page 66](#) describes the operations exposed by the MBean for a transport connector.

Table 9.6. Connector MBean Operations

Operation	Description
<code>void start();</code>	Starts the transport connector so that it is ready to receive connections from clients.
<code>void stop();</code>	Closes the transport connection and disconnects all connected clients.
<code>int connectionCount();</code>	Returns the number of open connections using the connector.
<code>void enableStatistics();</code>	Enables statistics collection for the connector.

Operation	Description
<code>void resetStatistics();</code>	Resets the statistics collected for the connector.
<code>void disableStatistics();</code>	Deactivates the collection of statistics for the connector.

Network connector actions

[Table 9.7 on page 67](#) describes the operations exposed by the MBean for a network connector.

Table 9.7. Network Connector MBean Operations

Operation	Description
<code>void start();</code>	Starts the network connector so that it is ready to communicate with other brokers in a network of brokers.
<code>void stop();</code>	Closes the network connection and disconnects the broker from any brokers that used the network connector to form a network of brokers.

Queue actions

[Table 9.8 on page 67](#) describes the operations exposed by the MBean for a queue destination.

Table 9.8. Queue MBean Operations

Operation	Description
<code>CompositeData getMessage(String messageId);</code>	Returns the specified message from the queue without moving the message cursor.
<code>void purge();</code>	Deletes all of the messages from the queue.
<code>boolean removeMessage(String messageId);</code>	Deletes the specified message from the queue.
<code>int removeMatchingMessages(String selector);</code>	Deletes the messages matching the selector from the queue and returns the number of messages deleted.
<code>int removeMatchingMessages(String selector, int maxMessages);</code>	Deletes up to the maximum number of messages that match the selector and returns the number of messages deleted.
<code>boolean copyMessageTo(String messageId, String destination);</code>	Copies the specified message to a new destination.

Operation	Description
<code>int copyMatchingMessagesTo(String selector, String destination);</code>	Copies the messages matching the selector and returns the number of messages copied.
<code>int copyMatchingMessagesTo(String selector, String destination, int maxMessages);</code>	Copies up to the maximum number of messages that match the selector and returns the number of messages copied.
<code>boolean moveMessageTo(String messageId, String destination);</code>	Moves the specified message to a new destination.
<code>int moveMatchingMessagesTo(String selector, String destination);</code>	Moves the messages matching the selector and returns the number of messages moved.
<code>int moveMatchingMessagesTo(String selector, String destination, int maxMessages);</code>	Moves up to the maximum number of messages that match the selector and returns the number of messages moved.
<code>boolean retryMessage(String messageId);</code>	Moves the specified message back to its original destination.
<code>int cursorSize();</code>	Returns the number of messages available to be paged in by the cursor.
<code>boolean doesCursorHaveMessagesBuffered();</code>	Returns <code>true</code> if the cursor has buffered messages to be delivered.
<code>boolean doesCursorHaveSpace();</code>	Returns <code>true</code> if the cursor has memory space available.
<code>CompositeData[] browse();</code>	Returns all messages in the queue, without changing the cursor, as an array.
<code>CompositeData[] browse(String selector);</code>	Returns all messages in the queue that match the selector, without changing the cursor, as an array.
<code>TabularData browseAsTable(String selector);</code>	Returns all messages in the queue that match the selector, without changing the cursor, as a table.
<code>TabularData browseAsTable();</code>	Returns all messages in the queue, without changing the cursor, as a table.
<code>void resetStatistics();</code>	Resets the statistics collected for the queue.
<code>java.util.List browseMessages(String selector);</code>	Returns all messages in the queue that match the selector, without changing the cursor, as a list.

Operation	Description
<code>java.util.List browseMessages();</code>	Returns all messages in the queue, without changing the cursor, as a list.
<code>String sendTextMessage(String body, String username, String password);</code>	Send a text message to a secure queue.
<code>String sendTextMessage(String body);</code>	Send a text message to a queue.

Topic actions

[Table 9.9 on page 69](#) describes the operations exposed by the MBean for a topic destination.

Table 9.9. Topic MBean Operations

Operation	Description
<code>CompositeData[] browse();</code>	Returns all messages in the topic as an array.
<code>CompositeData[] browse(String selector);</code>	Returns all messages in the topic that match the selector as an array.
<code>TabularData browseAsTable(String selector);</code>	Returns all messages in the topic that match the selector as a table.
<code>TabularData browseAsTable();</code>	Returns all messages in the topic as a table.
<code>void resetStatistics();</code>	Resets the statistics collected for the queue.
<code>java.util.List browseMessages(String selector);</code>	Returns all messages in the topic that match the selector as a list.
<code>java.util.List browseMessages();</code>	Returns all messages in the topic as a list.
<code>String sendTextMessage(String body, String username, String password);</code>	Send a text message to a secure topic.
<code>String sendTextMessage(String body);</code>	Send a text message to a topic.

Subscription actions

[Table 9.10 on page 70](#) describes the operations exposed by the MBean for a durable subscription.

Table 9.10. Subscription MBean Operations

Operation	Description
<code>void destroy();</code>	Destroys the subscription.
<code>CompositeData[] browse();</code>	Returns all messages waiting for the subscriber.
<code>TabularData browseAsTable();</code>	Returns all messages waiting for the subscriber.
<code>int cursorSize();</code>	Returns the number of messages available to be paged in by the cursor.
<code>boolean doesCursorHaveMessagesBuffered();</code>	Returns <code>true</code> if the cursor has buffered messages to be delivered.
<code>boolean doesCursorHaveSpace();</code>	Returns <code>true</code> if the cursor has memory space available.
<code>boolean isMatchingQueue(String queueName);</code>	Returns <code>true</code> if this subscription matches the given queue name.
<code>boolean isMatchingTopic(String topicName);</code>	Returns <code>true</code> if this subscription matches the given topic name.

Chapter 10. Applying Patches

Fuse MQ Enterprise supports incremental patching. FuseSource will supply you with easy to install patches that only make targeted changes to a deployed broker.

Index

A

- Active, 64
- activemq.xml, 14
- administration client
 - running, 43
- administration console
 - port number, 23
 - securing, 24
 - url, 23
- AverageEnqueueTime, 63

B

- BlockedProducerWarningInterval, 62
- broker
 - addConnector, 65
 - addNetworkConnector, 65
 - addQueue, 66
 - addTopic, 66
 - createDurableSubscriber, 66
 - destroyDurableSubscriber, 66
 - disableStatistics, 65
 - enableStatistics, 65
 - gc, 66
 - reloadLog4jProperties, 66
 - removeConnector, 65
 - removeNetworkConnector, 66
 - removeQueue, 66
 - removeTopic, 66
 - resetStatistics, 65
 - start, 65
 - stop, 65
 - stopGracefully, 65
 - terminateJVM, 66
 - useJmx, 60
- BrokerId, 62
- BrokerName, 62
- BrokerVersion, 62

C

- client, 43
- command console
 - getting help, 44
 - remote access, 45
- config shell, 14
- connector
 - connectionCount, 66
 - disableStatistics, 67
 - enableStatistics, 66
 - resetStatistics, 67
 - start, 66
 - stop, 66
- connectorHost, 61
- connectorPath, 61
- connectorPort, 61
- console
 - config shell, 14
- console mode
 - starting, 39
 - stopping, 48
- ConsumerCount, 63
- createConnector, 61
- createMBeanServer, 61
- CursorFull, 63
- CursorMemoryUsage, 63
- CursorPercentUsage, 63

D

- daemon mode
 - starting, 40
 - stopping, 48
- DataDirectory, 62
- DequeueCount, 63
- DequeueCounter, 64
- DispatchCount, 63
- DispatchedCounter, 64
- DispatchedQueueSize, 64

E

- embedded console
 - disabling, 25

EnqueueCount, 63
EnqueueCounter, 63
ExpiredCount, 63

F

fabric
 adding the Web console, 22
 profiles, 16
 starting a broker, 40
 stopping a broker, 51
 versions, 16
fabric:container-connect, 45
fabric:container-start, 40
fabric:container-stop, 51
features:install, 28
Fuse HQ, 10
Fuse Management Console, 10
fusemq, 39

I

InFlightCount, 63

J

jconsole, 10
JMX
 disabling, 60
jmxDomainName, 61

K

KARAF_BASE, 32
KARAF_DATA, 32
KARAF_HOME, 32

L

logging
 changing log level, 54
 configuration files, 54
 console commands, 56-57
 viewing as text, 56
 viewing in an editor, 56
 viewing in the console, 56

viewing with the admin client, 57

M

managementContext, 60
 connectorHost, 61
 connectorPath, 61
 connectorPort, 61
 createConnector, 61
 createMBeanServer, 61
 jmxDomainName, 61
 rmiServerPort, 61
 useMBeanServer, 61
MaxEnqueueTime, 63
MaximumPendingMessageLimit, 64
MaxPageSize, 63
MemoryLimit, 62-63
MemoryPercentageUsed, 62-63
MemoryUsagePortion, 63
MessageCountAwaitingAcknowledge, 64
MinEnqueueTime, 63
mq-web-console, 22

N

network connector
 start, 67
 stop, 67

O

org.apache.karaf.log.cfg, 54
org.ops4j.pax.logging.cfg, 54
org.ops4j.pax.logging.DefaultServiceLog.level, 54
osgi:shutdown, 50

P

PendingQueueSize, 64
persistent identifier, 13
PID, 13
PrefetchSize, 64
ProducerCount, 63

Q

queue

- browse, 68
- browseAsTable, 68
- browseMessages, 68-69
- copyMatchingMessagesTo, 68
- copyMessageTo, 67
- cursorSize, 68
- doesCursorHaveMessagesBuffered, 68
- doesCursorHaveSpace, 68
- getMessage, 67
- moveMatchingMessagesTo, 68
- moveMessageTo, 68
- purge, 67
- removeMatchingMessages, 67
- removeMessage, 67
- resetStatistics, 68
- retryMessage, 68
- sendTextMessage, 69

QueueSize, 63

R

- rmiServerPort, 61
- routine tasks, 9
- routing console
 - securing, 24

S

- service wrapper
 - classpath, 33
 - generating, 29
 - JMX configuration, 34
 - JVM properties, 33
 - logging, 34
- shutdown, 48
- ssh:ssh, 45, 50
- standalone broker
 - messaging configuration, 14
 - runtime configuration, 14
- start, 40
- stop, 48
- StoreLimit, 62

StorePercentageUsed, 62

subscription

- browse, 70
- browseAsTable, 70
- cursorSize, 70
- destory, 70
- doesCursorHaveMessagesBuffered, 70
- doesCursorHaveSpace, 70
- isMatchingQueue, 70
- isMatchingTopic, 70

system service

- Redhat, 36
- Ubuntu, 36
- Windows, 36

system.properties, 54

T

- TempLimit, 62
- TempPercentageUsed, 62
- tooling, 10
- topic
 - browse, 69
 - browseAsTable, 69
 - browseMessages, 69
 - resetStatistics, 69
 - sendTextMessage, 69
- TotalConsumerCount, 62
- TotalDequeueCount, 62
- TotalEnqueueCount, 62
- TotalMessageCount, 62
- TotalProducerCount, 62

U

- useJmx, 60
- useMBeanServer, 61

V

- VisualVM, 10

W

- Web console
 - accessing, 23

- adding to a fabric, 22
- clusters, 26
- port number, 23
- securing, 24
- webconsole.jms.url, 25-26
- webconsole.jmx.password, 25
- webconsole.jmx.url, 25-26
- webconsole.jmx.user, 25
- wrapper:install, 29